Contech: A Tool for Analyzing Parallel Programs

Brian P. Railing Eric R. Hein Phillip Vassenkov Thomas M. Conte

Georgia Institute of Technology (brian.railing, ehein6, pvassenkov3, conte @gatech.edu)

I. Introduction

The performance of emerging multicore processors can only be fully utilized by optimized, well-designed parallel programs. A number of tools exist to analyze the parallel programs and provide guidance to the programmer. Additional tools exist to support computer architects in their efforts to better support parallel programs in hardware. Each of these prior tools was designed to address one facet of a program.

The multicore era will require finding new aspects of the program for the compiler and architecture to exploit. Rather than constructing a targeted approach to analyzing an aspect, we propose a framework and representation that would support a variety of existing analysis types, such that combinations toward new aspects would be possible. Existing approaches were either so general that they could be a basis for our framework, or too specialized to an aspect or class of problems. To coordinate a variety of analyses, each needs to be applied to a common representation.

We demonstrate how our rich analysis framework allows a substantial reconstruction of a parallel program for analysis purposes without the necessity of repeated simulations. This analysis framework combines separate functionalities present in several prior projects. It is possible to then correlate between analyses and compose new analyses from prior pieces.

The analysis framework rests on a special program trace representation of interconnected program tasks, termed the task graph. Each task contains three streams of in-

formation: the instructions executed, the memory locations accessed, and the explicit scheduling dependencies between parallel components of the program. Our framework uses the scheduling constraints as a partial order of tasks, giving a graph representing the program's execution.

The task graph representation of a program has several advantages. First, a Contech task graph is independent of the threading library that is used to implement synchronizing actions; OS-specific implementation details have also been abstracted away. A Contech task graph is also architecture independent; for example, there is no ISA-specific information in the task graph. We believe that unifying a memory trace with basic block information and cross-thread dependencies will enable deeper analyses than can be carried out on an instruction or memory trace.

II. Task Graph

A shared-memory parallel program consists of more than one thread executing simultaneously. While the threads share the same virtual address space, they each maintain separate stacks and execute potentially different instruction streams. Pieces of work in task-oriented languages such as Cilk maintain similar state. We generalize the different notions of threading in these models by using the term "execution context".

We define the term "event" to represent a grouping of actions that are taken by an execution context. There are two basic kinds of events. The first is the execution of a basic block of instructions. The second is a call to a synchronizing action in a threading library. We classify all synchronization into four core types of synchronizing events: Create, Join, Sync, and Barrier. Each synchronizing event implies certain "sync dependencies" on other synchronizing events. Shown in Table 1, these events have a commonality across threading platforms, whether the platform is, for example, Pthreads, Cilk++[1], or OpenMP.

Table 1 - Example Mapping of Contech Events to Other Constructs

Contech	**Pthreads**	**Cilk**	**OpenMP**
Create	pthread_create	Cilk_spawn Cilk_for	omp task omp for
Join	pthread_join	Cilk_sync	omp taskwait
Sync	pthread_mutex	Cilk_lock	omp critical
Barrier	pthread_barrier	(implicit)	(implicit)

A Create event causes a new context to begin. Neither the parent nor the child can continue until the create has completed. A Join event causes one context to wait for another to finish. Sync events force an ordering between two contexts that access the same address. A context that executes a Sync on a given address must execute after the last

context that Sync'd on that same address. Barrier events specify that a given set of contexts wait for all others to execute a barrier event on the same address before continuing.

Instructions in a single thread are not specified to execute in a particular order, rather the task graph provides the program order of instructions from which the compiler and hardware can then reorder. Dependencies between these instructions restrict the set of possible reordering optimizations that the compiler or hardware can exploit. Instructions in different threads are inherently unordered with respect to each other, recall Lamport's partial ordering of events[2]. A correct parallel program uses synchronizing events to restrict the set of possible orderings of threads to those that preserve the intended dependencies between memory writes and reads in different threads.

We define a "task" as a sequence of basic block events executed by a single context. Synchronizing events mark the boundaries between tasks. Since a context executes code sequentially, we may assume that each synchronizing event is dependent on the task that executed before it, and is itself a predecessor for the task that the context will execute next. We call these "seq dependencies" since they arise from the sequential nature of a single thread's execution.

Together, "seq" and "sync" dependencies establish a partial order on the execution of tasks. This forms the foundation for communication between tasks. Two tasks communicate when one is ordered before the other, the first executes a write, and the second executes a read to the same address. Data races occur when tasks that are not ordered by such dependencies try to communicate.

Building upon these definitions, the execution of a parallel program can be characterized by a "task graph" which consists of tasks, synchronizing events, and the de-

pendencies between them. Tasks are characterized by the instructions they execute, the data they touch, and their predecessor tasks.

III. Design Tradeoffs

From a high level perspective, Contech has three key purposes: collecting runtime data, refining and aggregating the collected data, and enabling analyses on the data. There are several possible design architectures enabling different ways to accomplish these goals, each with their pros and cons. In subjectively analyzing these approaches, metrics are defined on which to evaluate them: cost of implementation, information flexibility and robustness, performance cost, and performance scalability. The cost of implementation describes how many man-hours are required to implement a solution using a certain architecture to achieve desired functionality. This is generally the initial time investment that the tool designers must account for. The next metric is a combination of information collecting flexibility and information robustness. It qualifies design rigidity, how sensitive the cost of implementation is to changes in functionality requirements (i.e., flexibility) and the quality of the information collected (i.e., robustness). The approaches still have certain overheads, so the performance cost metric gauges the cost of the user's source code executing and yielding analysis results and fruitful data. Performance scalability is another metric that accounts for how much the performance cost increases with the complexity of user requirements. Together, these four metrics help compare the various design approaches for the Contech system.

One approach to meeting the system design goals is via architectural simulation. In general, architectural simulators consist of an instruction fetching frontend and an instruction simulating backend. The complexity of a dynamic instruction fetching unit in most modern architectures along with the numerous amount of events that need to be simulated on the backend reduce this approach's ratings across both the cost of imple-

mentation and performance cost metrics. Contech and SimpleScalar both employ execution driven simulation models because more information can be gleaned about the programmer's intent from the binary rather than a trace. Although SimpleScalar has a tuning parameter responsible for the tradeoff between performance and accuracy, performance is only bounded by three to four orders of magnitude slower than native execution[3].

In order to increase the performance of the architectural simulator, one must reduce the accuracy of simulation detail by replacing detailed architectural components with behavior-emulating black boxes. This loss is detail can greatly harm the detection of unknown hardware behavior side effects. In addition, it is also not very flexible, as new user requirements would require simulator modification. However, gem5 offers flexibility by removing the burden of choosing amongst implementation technologies and simply supporting as many as possible. Performance scalability is not a strongpoint for this approach because increase in user requirements puts a high degree of strain on the rest of the system, as many components are interrelated.

Another approach to meeting the design goals is via binary instrumentation. This entails running software that inserts event recording function calls among binary instructions at runtime. Intel's Pin is a popular binary instrumentation framework that could have been used as a means for implementing Contech, including Pin's emphasis on architectural independence. The Pin framework allows users to create *pintools* to plug into Intel's proprietary instrumentation framework with the intent of controlling how a binary is instrumented and what analysis is performed. This enables a low cost of implementation when collecting program runtime behavior. The framework allows users to specify at which granularity to instrument the binary: instruction, basic block, function level, and

more. The Pin framework will then call the user's *pintool* functions at the appropriate times, performing the necessary information collection, aggregation, and analysis; however, Pin differs sharply in that Contech supports only offline analyses. The performance cost can be low, but tends to scale poorly, especially if analysis is done as part of an online algorithm [4]. Pin is fairly flexible about what types of information it collects; however, the more information that is exposed to the *pintool*, the more of a performance hit the program will take.

The third method examined was a compiler pass based approach to source code instrumentation. Generally, writing compiler passes has an enormous cost of implementation because of their correctness requirements and complexity. A compiler pass has the greatest degree of information flexibility and robustness because it is the component that sits between software and bare metal hardware. The performance cost overhead of compiler passes is already low, but with the added benefit of iterative compiler optimization even lower overheads can be achieved. The performance scalability of this approach is quite competitive. The compiler pass would add functions to execute at specific points in the code. These functions do not affect each other's performance. By comparison to the architectural simulation approach, where additional complexity in one component can put strain on all other parts of the system, a compiler based approach has a lower performance scalability rating because of the independence of the instrumented functions.

Table 2 – Summary of the tradeoffs amongst the approaches

	Architectural Simulation	Binary Instrumentation	Source Instrumentation
Cost of Implementation	* *	* *	* * *
Information Flexibility & Robustness	* *	*	*
Performance Cost	* * *	* *	*
Performance Scalability	* *	* *	*

After analyzing the various possible approaches in designing such a system, it was decided that the best approach to use was one closely tied to the compiler approach. Table 2 summarizes the qualitative design tradeoffs for each approach. One of the goals is to have the lowest performance cost, but that was a common problem in modern program analysis frameworks: users have to wait excruciatingly painful amounts of time in order to get data about the nature of program execution. Another goal was to get the most detailed and appropriate information to enable the most powerful analyses. Performance insensitivity to user requirements, defined as performance scalability, was also a goal. All of these can be bought with the high implementation cost that comes with a compiler based approach.

IV. Frontend Architecture

The Contech frontend is the section of the framework responsible for taking source code, instrumenting source code with calls to the Contech runtime library, applying the standard compiler passes and optimizations, and producing an instrumented binary executable. The primary Contech front-end uses the LLVM compiler framework to make the required changes to intermediate representation during compilation to include the appropriate instrumentation to the data required to create a task graph.

Contech is interested in collecting two big picture types of information, common to all multicore architectures: memory access behavior and parallel execution ordering constraints. The event types of interest were gathered with the intent of making Contech cross platform, so events representing common behavior/functionality regardless of programming paradigm or source language are instrumented.

Main

The instrumentation attempts to leave as much of the code unchanged as possible. One important exception is the program's 'main' function (in the context of C/C++ programs). The program's 'main' is renamed and called from a 'main' method introduced by Contech (located in the Contech runtime library, linked in during compilation), which is called by the system runtime (libc). This new main function is responsible for performing the necessary instrumentation framework setup, such as allocating resources for components of the buffering architecture (discussed later), and cleanup/teardown. Contech currently assumes that 'main' is the entry point to the program, although it is a simple problem to handle program entry points with other names. This eliminates the burden of having to insert special Contech initialization instructions into the code before the "start" of

the program. There is further code in the framework that handles routines that execute for the “start”, like constructors for static classes.

Parallel Programming Primitives

A majority of the other instrumentation event types are related to the synchronization primitives discussed earlier. When a program spawns a new software context, commonly known as a thread, this event is labeled as a ‘TASK_CREATE’ event. Related to this event is the ‘TASK_JOIN’ event, which occurs when a program either terminates one of its threads or when a thread joins into another thread. Creating and destroying threads of execution is a notion common many parallel programs. Another group of similar events are the actual synchronization primitives, namely ‘SYNC_AQUIRE’, ‘SYNC_RELEASE’, and ‘BARRIER_WAIT’. The two sync events (acquire and release) correspond to the acquisition and release of a synchronization variable: a lock, a semaphore, or some other type of synchronization variable. Likewise, in order to capture multiple threads waiting on barrier synchronization primitives, barrier blocking operations must be captured. Capturing these parallel programming primitives is essential to displaying the ordering constraints amongst the various parallel portions of the program.

Memory Operations

Another essential aspect to capturing the program behavior, aside from the ordering constraints captured amongst parallel sections of code, is the memory access pattern of the program itself. There are a couple types of memory events that are instrumented: individual instructions and calls to memory transfer functions. Since Contech has access to LLVM’s IR code, it can instrument individual load and store instructions, as well as function calls to memory-related functions such as memcpy, malloc, or free.

Contech chooses to express program behavior using a handful of parallel programming primitives. It would not be too difficult to add new types of events for Contech to handle. In order to support different parallel programming paradigms, there is a decoupled mapping of event type and a function name which triggers said event. The functions in this function map are to be treated as special and the function map serves as the "entry point" for adding special event types throughout the rest of the Contech framework.

Frontend Workflow

Initially the Contech compiler pass performs initialization and required setup such as setting up resources and getting runtime callback functions ready for instrumentation. After setup has completed, the actual instrumentation may begin. LLVM's definition of a basic block allows for function calls inside of a basic block (as long as they return into the same block), leaning closer to the definition of an extended basic block. All of the basic blocks must be 'normalized' to have at most one function call per basic block, modifying the definition to be closer to the original definition of a basic block. Splitting basic blocks at function calls allows us to reason about per basic block memory behavior in a way that eliminates the possibility of outside basic blocks from influencing (loading or storing data into) external memory addresses.

Once the blocks are normalized, block level instrumentation may begin. This entails examining all of the instructions in a basic block and performing different actions based on the kind of instruction examined. Most of the actions require recording information. Writing out to file every time an event occurs would severely slow down the program, so to combat this, the data is recorded into an in-memory buffer, managed by the mentioned buffering architecture, in a background thread. The general approach for re-

cording information is to insert a function call into the LLVM IR, to call code in the ct_runtime library (which is linked into the binary).

Memory Access Behavior

Reading and writing data from memory requires a way to address individual memory locations, a characteristic common to most computers' architectures, and consequently the programs that run on them. Recording this information is crucial to understanding how data moves around in a program. In the event of any type of memory transaction operations encountered, whether it is a load, store, dynamic memory allocation, or bulk memory transfer (memove, memcpy, etc...), a callback to the ct_runtime library is inserted in order to record the memory operation. Storing the record in the buffer on every occurrence of a memory event allows Contech to glean a great amount of information (addresses involved, size of operations, length of time to process that instruction) while keeping runtime overhead low.

Ordering

Another aspect of program behavior common to any parallel execution architecture is the ordering with which serial segments of code are executed. The remaining (besides the memory functions mentioned) function calls instrumented by Contech are the sources of ordering constraints. In the event of a function call instruction, using the function map mentioned previously, the name of the function is mapped to the type of event encountered. This decouples functionality from source code constructs, allowing Contech to be extended onto other programming paradigms expressing similar core functionality.

Thread Create

When a program starts running, the first program thread executes instructions from the program's 'main'. Any additional threads of execution spawned from the initial or subsequently spawned threads must come from program instructions, either implicitly or explicitly inserted by the programmer. When such an instruction is encountered, it is labeled as a 'TASK_CREATE' event. The function call that spawns a thread is replaced by a call to the ct_runtime library. The ct_runtime library function acts as a wrapper for thread spawning behavior. This enables Contech to perform the necessary internal setup for thread creation, which includes generating unique internal thread IDs, resolving timing information, preparing the buffering component for this thread, and then leveraging said component to store the 'TASK_CREATE' event.

When Contech reasons about threads of execution from a thread-level perspective, several issues become apparent, which have low cost solutions in place. Threads are scheduled to run on a core by the process dispatcher of an operating system. For a variety of reasons related to fairness, threads can have their execution preempted, state saved, and scheduled to run on a potentially different hardware core at a later time. The event file references events originating from certain threads, and in order to consistently refer to the same thread, an ID (maintained by the ct_runtime) is assigned to each thread, termed the 'contech_id'. Each event has the relevant 'contech_id' recorded as well.

The event file, being a trace, must contain timing information as well, so that consumers can reason about the relative length of wall-clock time spent performing certain actions. The goal is to obtain timing information with as low of a cost as possible. To do this, Contech needed to poll a computer's notion of time; however, the cheapest way to find this information is through the time stamp counter, a counter for the number of cy-

cles (since machine reset). For fast access, each processor has its own counter. With dynamic frequency and voltage scaling being a popular means of throttling processors, there is a possibility of clock skew in thread timing information. Another threat to timing accuracy occurs in the event that a thread is preempted and scheduled to run on a different core. In that case, the timing information for events inside a single Contech could be inaccurate. With respect to these issues and the purity of the task graph representation, the low cost timing information is included for backend analysis, but Contech backends should not rely on the accuracy of timing information throughout the event file, rather on ordering information among synchronization primitives. A simple algorithm exists in the frontend to estimate the skew, in order to reduce the inaccuracies observed by backends.

Thread Join

At some point in a parallel program, it may be desired to ensure that a thread has completed execution before progressing further in the program. One example of this behavior occurs towards the end of the program. Since it is usually undesirable for the main thread of execution to terminate itself (and consequently all of its spawned threads), this mechanism enables the program to deal with mentioned problem. When one thread waits for the termination of another thread, this is referred to as a 'TASK_JOIN' event. Such an event, when encountered, is forwarded to the buffering component for eventual output to the event file.

Sync Acquire

Managing critical sections and the number of threads allowed to execute some code is done usually through the use of synchronization (sync) variables (and barriers, but differences are to be discussed later). When the instrumented program is executed and a

thread acquires a sync variable, the type of event recorded will be ‘SYNC’. Further processing reveals whether the ‘SYNC’ event represents the acquisition or release of a synchronization variable.

Sync Release

The amount of time spent holding a sync variable while in a critical section can vary, so the event file must note not only the acquisition, but also the release of a lock variable. Sync variable release is noted in the event file with a ‘SYNC’, but the distinction of acquisition from release is determined later.

Barrier

When a program needs to have several threads all reach a section before proceeding forward, this is traditionally done with a barrier or similar structure. This event is marked as a ‘BARRIER_WAIT’ in the event file.

Buffering Architecture

As the compiled program runs, Contech inserted instrumentation is executed and events are being written out to disk. The events written out to disk are in a certain order. There are two types of order preserved in the event file: partial ordering and semantic ordering. The partial ordering is present because events that are tagged with the same contech_id will come from the same thread of execution. Amongst all the events that originate from the same thread and share the same contech_id, there exists a total ordering. The semantic ordering refers to the assumption that the source code is correct and ensures events have a correct logical ordering between them (create & join, acquire & release). Several approaches were evaluated in order to finally conclude on the actual implementation.

When evaluating the various possible approaches, the metric used to quantify the fitness of an approach was its overhead in terms throughput, the overhead's impact on shared resources (and consequently the user program), and runtime footprint. Were each instrumented callback to ct_runtime write out to the file directly, this would have a great effect on the user program and have too much overhead. In order to reduce the impact of both these factors (respectively), events should be written out to file asynchronously and they should also be buffered/aggregated for batch writing. Ideally, the events would all be collected with minimal overhead at runtime, and then written out to file after the user program completes execution. The issue with this approach is that realistically, the space consumed by the buffers would grow too large; the program will crash once it fills the machine's main memory. To address this issue, the buffering component leverages periodic buffer flushing (i.e. writing out to file). Transmitting information primarily through the use of pointers further reduces the time spent copying data at runtime.

. To summarize the design of the Contech front-end, a program instrumented by Contech will have function calls inserted at certain key points to record events. When any new threads are spawned, Contech allocates a buffer chunk in thread local storage. For most cases, when control is transferred from the user's program to the ct_runtime an event is copied into the buffer. Since these buffers are of a fixed size, a mechanism must exist to ensure that they do not overflow. Contech inserts buffer fullness checks at certain points, such as at the end of every basic block. In order to reduce the impact of redundant checks, the Contech compiler pass utilizes a dominator tree to remove redundant basic block buffer size checks inside loops, placing a check only at the loop header. When a buffer is detected as full, it is queued to be written out to file by a background thread. The

user program thread then allocates an empty buffer from a pool of buffers. Once the background thread drains the full buffer, it is returned to the pool for reuse.

V. Middle Layer Architecture

While the event file is a complete trace of the events captured throughout program execution, it is not easily consumable in terms of information format. The next component in the Contech framework is termed the middle layer. It is the component responsible for aggregating events from the event file into discrete units of work (i.e., tasks) that may proceed in parallel and finally connecting them with ordering constraints to form a task graph. The goal of this format is to be able to describe parallel program behavior in a succinct format, in a manner that does not tie it to any specific computer architecture.

A task is similar to a basic block, except at a much higher abstraction level. A task is a serial piece of code (i.e., series of basic blocks) that is terminated by one of the fundamental parallel programming constructs, such as acquiring or releasing a synchronization variable, or creating a new thread. A task is comprised of the code that runs on a thread. Each task contains information to answer three questions: what code ran, what memory addresses were accessed, and what other tasks depended on this one. The first two questions are answered by a list of chronologically ordered *ct_action* structures that describe a list of basic blocks executed or memory operations performed, either via a sequence of loads and stores or grouped into a bulk memory operation. The chronological ordering of events within a task is guaranteed because the events occur sequentially in each logical program thread.

Besides the contents of tasks, the task graph also stores the edges in the graph between the tasks. These edges designate execution dependencies between tasks. The special tasks, shown earlier in Table 2, correspond to the synchronization primitives discussed earlier, i.e. *TASK_CREATE*, *TASK_JOIN*, *SYNC*, and *BARRIER_WAIT*. The pur-

pose of having these special tasks is to be able reason about the successor and predecessor tasks separately from regular tasks. The special tasks form ordering constraints among tasks. A simple example is *TASK_CREATE*: in the graph, a task entering a special *TASK_CREATE* task will then be split into two tasks. A more complicated example is the special task of *BARRIER_WAIT*. When a program contains a barrier for multiple threads, the corresponding tasks will have their successors be a single *BARRIER_WAIT* task, and the successors of the *BARRIER_WAIT* are all tasks that can then begin to execute once the necessary number of threads arrive at the barrier. There is a generalized statement that can be made about one or more tasks connected together: a task cannot begin execution until all of its predecessors have completed execution. One interesting corollary of this is that this property is transitive. Since a task graph is a partial ordering of tasks, a partial order's property of transitivity can be applied here. For some chain of tasks, it can safely be said which tasks can happen before others. If A's successor is B and B's successor is C then A and C have an ordering between them. The goal in designing the task graph layout was to capture a program trace that would enable reasoning about which pieces of work can or cannot execute in parallel, while striving to be free from hardware specific information. Once a trace is collected, all the information necessary to analyze program behavior offline is read.

The middle layer receives as input the event file, as it was output by running the instrumented user program. During initialization, one key data structure allocated is a list of tasks per thread. Events are bucketized into the appropriate task list and coalesced to form tasks. Since the event list is a chronological partial ordering, this property is also carried forth into the task graph. After one of the synchronization primitives is encoun-

tered, it signals that all the events pertaining to a certain task have been processed and that a new task may be started. These special events take on special meaning in the middle layer as well. After a *TASK_CREATE* event is encountered, it brings up the question of whether the event belongs to the parent task or the newly spawned child task. This is answered via a boolean array describing which threads have been already executing, the parents, and which ones are only beginning to execute, the children. The two types of *SYNC* events, *SYNC_ACQUIRE* and *SYNC_RELEASE*, require the middle layer to track lock ownership, at the address granularity, in order to create the association between a lock variable being locked and unlocked by a thread. *BARRIER_WAIT* event types also have their own technical difficulties. In order to handle the possibility of a barrier being reached at a specific address multiple times throughout program execution, the middle layer maintains a list of barrier coordination information amongst threads. The loop embedded barrier issue can arise because only a partial ordering of barrier events is guaranteed and every dynamic barrier instance corresponds to the same static barrier instance. It is guaranteed that all the arrivals into a dynamic instance of a barrier occur before any departures. No guarantee exists to prevent a thread's arrival at the next dynamic barrier instance before all threads have left the current barrier instance. Another one of the middle layer's responsibilities is recalculating the absolute clock cycle information to be relative to the start of the user program. Once the middle layer finishes processing through the event list, and constructing all of the tasks, they are then written out to disk. The bucketized nature of middle layer's algorithm allows for a deterministic ordering on the task graph output. Tasks are written to file chronologically, and in the event of an end time conflict among tasks, the tie-breaking rule is writing out in ascending *contech_id*

order. The serialization of tasks is handled by another library, known as *taskLib*. Among other things, it facilitates reading and writing task graph files, and provides an API for interacting with individual tasks.

VI. Backend Architecture

A Contech backend is modular piece of code that consumes a task graph and performs a transformation or an analysis on it. The types of backends have been generalized to three different types, labeled by the backend's type of output: modified task graph, complex format, or statistics. A backend that outputs a modified task graph could potentially be modifying the structure or contents of the task graph in order to gain further insight about program behavior or reducing noise. Other types of backends, which output other complex formats, transform the task graph into another type of format, as long as they describe program behavior to some degree. The third type of backend is one that produces statistics. These backends perform analysis on a task graph and output metrics to summarize what has been learned by the analysis. These backends are assisted by an API in *taskLib* which enables backends to read and write task graph files, along with accessing the different robust pools of information in a task, such as the streams of memory operations, basic blocks executed, or successors/predecessors of the task.

VII. Measurements

As with any instrumentation approach, Contech has certain overheads, both during runtime and during compile time. This section includes performance measurements from running the Parsec 3.0 benchmark suite[5] on a cluster of machines configured per Table 3, where each benchmark was run using the simmedium input set. Table 4 shows the time to compile the PARSEC benchmarks, as well as time to run each benchmark with and without the instrumentation. The time to compile each application is doubled, as two compiler passes are required for each source file. The runtime increases by 3x to 230x with the instrumentation; the flush time is the time after the benchmark has completed for the list of events to flush to disk. This runtime impact reflects characteristics of the program, as programs that are highly synchronous incur more overhead to maintain the orderings of events (see Table 3 for program characteristics). Other work has shown these slowdowns comparable for the level of instrumentation required using a framework like Pin [4].

Table 3 – Experimental system configuration

Processor Model	Intel Xeon X5670
# of Processors	2
Cores per Processors	6
Hyperthreading	2-way
Clock Speed	2.93 GHz
Last Level Cache	12 MB
Main Memory	47 GB

Table 4 - Time to Compile and Run Parsec and Splash2 with Clang and Contech

	Clang		Contech LLVM		Run Time
Benchmark	Compile	Run Time	Compile	Run+Flush Time	Slowdown
blackscholes	2.50	0.11	2.93	0.37+8.25	3.35
bodytrack	48.27	0.41	90.07	6.52+156.82	15.90
fluidanimate	3.89	1.00	8.42	115.34+207.66	115.34
swaptions	3.19	0.41	6.03	8.45+328.70	20.61
barnes	3.11	0.19	6.03	9.83+234.50	51.72
cholesky	5.21	0.11	13.16	3.44+79.40	31.31
fft	1.89	1.17	3.20	4.95+277.05	4.23
fmm	3.64	0.32	8.25	15.67+464.25	48.96
ocean_cp	4.11	0.67	13.20	11.92+871.80	17.79
ocean_ncp	3.26	0.92	8.81	11.94+720.93	12.97
radiosity	4.66	0.28	12.01	66.35+409.56	236.96
radix	1.90	0.27	3.73	2.62+205.96	9.72
water_spatial	3.67	0.48	9.14	20.14+680.56	41.96

VIII. Summary

This technical report details the purpose, internal mechanisms, and performance of a modular parallel program analysis framework termed Contech. Contech's goal is to instrument user source code to produce a robust trace, termed the taskgraph, containing the instructions executed, the memory locations accessed, and the explicit scheduling dependencies between parallel components of the program. Since Contech's frontend is an LLVM compiler pass, it gains LLVM's ability to consume source code written in different languages and utilizing various parallel programming paradigms. Contech's modular design naturally enables backends to perform multiple offline analyses of a taskgraph. Since Contech also stresses the importance of collecting architecturally independent information, the taskgraph captures the core program behavior without diluting the information with hardware specific behavior or artifacts.

-

References

[1] C. E. Leiserson, “The Cilk++ concurrency platform,” *Proc. 46th Annu. Des. Autom. Conf. ZZZ - DAC ’09*, p. 522, 2009.

[2] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” in *Communications of the ACM*, 1978, vol. 21, no. 7, pp. 558–565.

[3] S. Modeling, “SimpleScalar : An Infrastructure for Computer Developed to provide an infrastructure for simulation and architectural,” no. February, pp. 59–67, 2002.

[4] M. Bach, M. Charney, and R. Cohn, “Analyzing parallel programs with pin,” *Computer (Long. Beach. Calif).*, pp. 56–63, 2010.

[5] C. Bienia and K. Li, “Benchmarking Modern Multiprocessors,” 2011.